문자열

2020-01-19

이승진

학습목표

string

string 메소드

목차

[1. string 2](#_Toc36577898)

[1) 따옴표 2](#_Toc36577899)

[2) 문자 접근 3](#_Toc36577900)

[3) immutable string 3](#_Toc36577901)

[4) template literal 4](#_Toc36577902)

[2. string 메소드 5](#_Toc36577903)

[1) indexOf 메소드 5](#_Toc36577904)

[2) lastIndexOf 메소드 5](#_Toc36577905)

[3) search 메소드 6](#_Toc36577906)

[4) slice 메소드 6](#_Toc36577907)

[5) substring 메소드 7](#_Toc36577908)

[6) substr 메소드 7](#_Toc36577909)

[7) replace 메소드 8](#_Toc36577910)

[8) toLowerCase, toUpperCase 메소드 9](#_Toc36577911)

[9) concat 메소드 9](#_Toc36577912)

[10) trim 메소드 9](#_Toc36577913)

[11) charAt, charCodeAt 메소드 10](#_Toc36577914)

[12) split 메소드 11](#_Toc36577915)

[13) match 메소드 13](#_Toc36577916)

[3. 요약 14](#_Toc36577917)

[4. 부록 - 정규식 (regular expression) 16](#_Toc36577918)

[1) 정규식 특수 문자와 일반 문자 16](#_Toc36577919)

[2) [ ] 특수 문자 16](#_Toc36577920)

[3) . 특수 문자 17](#_Toc36577921)

[4) [^ ] 특수 문자 17](#_Toc36577922)

[5) ? 특수 문자 17](#_Toc36577923)

[6) \* 특수 문자 17](#_Toc36577924)

[7) + 특수 문자 18](#_Toc36577925)

[8) ^ 특수 문자 18](#_Toc36577926)

[9) $ 특수 문자 18](#_Toc36577927)

[10) {*n*} 특수 문자 18](#_Toc36577928)

[11) {*n,m*} 특수 문자 18](#_Toc36577929)

[12) ( ) 특수 문자 19](#_Toc36577930)

[13) | 특수 문자 19](#_Toc36577931)

[14) \ 특수 문자 19](#_Toc36577932)

[15) 연습문제 20](#_Toc36577933)

[5. 과제 21](#_Toc36577934)

[1) 모음 위치 (vowel.js) 21](#_Toc36577935)

[2) 공백 제거 (space.js) 21](#_Toc36577936)

[3) 쪼개기 (split.js) 21](#_Toc36577937)

# string

## 따옴표

javascript 문자열은 " 문자로 묶어도 되고, ' 문자로 묶어도 된다.

### string1.js

|  |
| --- |
| let s1 = "hello";  let s2 = 'world';  console.log("%s %s", s1, s2); |

출력

|  |
| --- |
| hello world |

따옴표 문자를 출력하는 방법은 다음과 같다.

### string2.js

|  |
| --- |
| let s1 = '"hello world"';  let s2 = "'hello world'";  let s3 = "\"hello world\"";  let s4 = '\'hello world\'';  console.log(s1);  console.log(s2);  console.log(s3);  console.log(s4); |

출력

|  |
| --- |
| "hello world"  'hello world'  "hello world"  'hello world' |

문자열 내부에 " 문자가 포함되어 있다면, 그 문자열은 ' 문자로 묵어야 한다. '"hello world"'

문자열 내부에 ' 문자가 포함되어 있다면, 그 문자열은 " 문자로 묵어야 한다. "'hello world'"

문자열 내부의 " 문자나, ' 문자 앞에 \ 문자를 붙였다면, (escape sequence)

그 문자열을 묶는 문자는 아무 거나 선택할 수 있다.

## 문자 접근

### string3.js

|  |
| --- |
| let s = "hello world";  for (let i = 0; i < s.length; ++i)      console.log(s[i]); |

출력

|  |
| --- |
| h  e  l  l  o    w  o  r  l  d |

[ ] 연산자를 사용하여 문자열의 문자를 얻을 수 있다.

## immutable string

### string4.js

|  |
| --- |
| let s = "hello world";  console.log(s[0]);  s[0] = 'H';  console.log(s[0]); |

출력

|  |
| --- |
| h  h |

javascript 문자열은 immutable 하다.

문자열이 생성된 후, 그 내용이 변경될 수 없다.

문자열 메소드들은 원본 문자열을 변경하는 것이 아니고, 새 문자열을 생성해서 리턴한다.

## template literal

### template1.js

|  |
| --- |
| let first = 'Jane';  let last = 'Doe';  let s = `Hello ${first} ${last}!`  console.log(s); |

` 따옴표를 사용한 문자열에 ${ 표현식 } 형태로 구현하여, 표현식의 값이 문자열에 삽입된다.

키보드 오른쪽 위의 ` 따옴표 문자를 사용해야 한다.

# string 메소드

## indexOf 메소드

### 문자열.indexOf(부분\_문자열, 시작\_위치)

**문자열**에서 **부분\_문자열**을 찾아서, 찾은 위치(index)를 리턴한다.

**시작\_위치**에서부터 뒤쪽으로(순방향으로) 찾기 시작한다.

**시작\_위치** 파라미터가 생략된 경우 디폴트 값은 0 이다.

찾지 못할 경우 리턴값은 -1 이다.

### indexOf.js

|  |
| --- |
| let s = "one two one two";  console.log(s.indexOf("two"));  console.log(s.indexOf("two", 5));  console.log(s.indexOf("TWO")); |

출력

|  |
| --- |
| 4  12  -1 |

## lastIndexOf 메소드

### 문자열.lastIndexOf(부분\_문자열, 시작\_위치)

**문자열**에서 **부분\_문자열**을 찾아서, 찾은 위치(index)를 리턴한다.

**시작\_위치**에서부터 앞쪽으로(뒷방향으로) 찾기 시작한다.

**시작\_위치** 파라미터가 생략된 경우 디폴트 값은 문자열의 끝 위치이다.

찾지 못할 경우 리턴값은 -1 이다.

### lastIndexOf.js

|  |
| --- |
| let s = "one two one two";  console.log(s.lastIndexOf("two"));  console.log(s.lastIndexOf("two", 11));  console.log(s.indexOf("TWO")); |

출력

|  |
| --- |
| 12  4  -1 |

## search 메소드

### 문자열.search(/정규식/)

**문자열**에서 **정규식**을 찾아서, 찾은 위치(index)를 리턴한다.

javascript 언어에서 정규식은 **/정규식/** 형태이다.

### search.js

|  |
| --- |
| let s = "hello World";  console.log(s.search(/[A-Z]/)); |

출력

|  |
| --- |
| 6 |

### 정규식(regular expression)

정규식은, 패턴으로 문자열을 찾기 위한 문자열 패턴 표현식이다.

[A-Z] 정규식은 A부터 Z까지 문자 중 아무 한 문자를 의미한다.

즉 대문자 한 개와 일치하는 정규식이다.

정규식에 대한 설명은 부록에서... (부록은 학습 목표에 포함 안됨)

## slice 메소드

### 문자열.slice(시작\_위치, 끝\_위치)

문자열에서 **시작\_위치**에서 **끝\_위치** 사이의 부분 문자열을 리턴한다. (시작\_위치 <= 위치 < 끝\_위치)

원본 문자열은 변하지 않는다. (immutable)

**끝\_위치** 파라미터를 생략한 경우 디폴트 값은 문자열의 길이이다.

**시작\_위치** 값이 음수이면, 문자열 끝에서 역방향 인덱스이다.

### slice.js

|  |
| --- |
| let s = "abcdefgh";  console.log(s.slice(2, 4));  console.log(s.slice(2));  console.log(s.slice(-3)); |

출력

|  |
| --- |
| cd  cdefgh  fgh |

## substring 메소드

substring 메소드는 slice 메소드와 유사하다.

차이점은, substring 메소드는 음수 파라미터를 사용할 수 없다는 점이다.

### substring.js

|  |
| --- |
| let s = "abcdefgh";  console.log(s.substring(2, 4));  console.log(s.substring(2));  console.log(s.substring(s.length - 3)); |

출력

|  |
| --- |
| cd  cdefgh  fgh |

## substr 메소드

substr 메소드는 slice 메소드와 유사하다.

차이점은, substr 메소드의 두 번째 파라미터는 부분 문자열의 길이라는 점이다.

### substr.js

|  |
| --- |
| let s = "abcdefgh";  console.log(s.substr(2, **2**));  console.log(s.substr(2));  console.log(s.substr(-3)); |

출력

|  |
| --- |
| cd  cdefgh  fgh |

## replace 메소드

### 문자열.replace(부분\_문자열, 치환할\_문자열)

replace 메소드는, 문자열에서 **부분\_문자열**을 찾아서 **치환할\_문자열**로 치환한다.

일치하는 부분 문자열이 여러 개 있어도, 첫 번째 부분 문자열만 치환된다.

**부분\_문자열**에 정규식을 사용할 수 있다.

원본 문자열은 수정되지 않고, 치환된 새 문자열이 만들어져서 리턴된다.

### replace1.js

|  |
| --- |
| let s = "hello world";  console.log(s.replace("o", "O"));  console.log(s); |

출력

|  |
| --- |
| hellO world  hello world |

원본 문자열, 즉 s 값은 변경되지 않았음에 주목하자. (immutable)

첫 번째 "o" 부분만 변경되었음에 주목하자.

### replace2.js

|  |
| --- |
| let s = "hello world";  console.log(s.replace(/o/g, "O"));  console.log(s); |

출력

|  |
| --- |
| hellO wOrld  hello world |

"o" 부분이 전부 변경되었음에 주목하자.

/정규식/g

g는, 일치하는 부분 문자열들을 전부 치환하라는 옵션이다.

/정규식/i

i는, 문자열을 비교할 때, 대소문자를 무시하라는 옵션이다.

## toLowerCase, toUpperCase 메소드

toLowerCase 메소드 - 모든 문자를 소문자로 변경한 새 문자열을 리턴한다.

toUpperCase 메소드 - 모든 문자를 대문자로 변경한 새 문자열을 리턴한다.

원본 문자열은 변경되지 않는다.

### toLowerCase.js

|  |
| --- |
| let s = "Hello World";  console.log(s.toLowerCase());  console.log(s.toUpperCase()); |

출력

|  |
| --- |
| hello world  HELLO WORLD |

## concat 메소드

파라미터 문자열들을 결합한 새 문자열을 리턴한다.

원본 문자열은 변경되지 않는다.

### concat.js

|  |
| --- |
| let s1 = "hello";  let s2 = "world";  console.log(s1.concat(" ", s2, "!"));  console.log(s1 + " " + s2 + "!"); |

출력

|  |
| --- |
| hello world!  hello world! |

## trim 메소드

문자열 앞뒤의 공백을 제거한 새 문자열을 리턴한다.

원본 문자열은 변경되지 않는다.

### trim.js

|  |
| --- |
| let s = "  hello world  ";  console.log("[%s]", s.trim());  console.log("[%s]", s); |

출력

|  |
| --- |
| [hello world]  [ hello world ] |

## charAt, charCodeAt 메소드

### 문자열[위치]

문자열에서 위치(index)의 문자를 리턴한다.

### 문자열.chatAt(위치)

문자열에서 위치(index)의 문자를 리턴한다.

### 문자열.chatCodeAt(위치)

문자열에서 위치(index)의 문자의 코드 값을 리턴한다.

### chatAt.js

|  |
| --- |
| let s = "hello world";  for (let i = 0; i < s.length; ++i)    console.log("%s %s %d", s[i], s.charAt(i), s.charCodeAt(i)); |

출력

|  |
| --- |
| h h 104  e e 101  l l 108  l l 108  o o 111  32  w w 119  o o 111  r r 114  l l 108  d d 100 |

|  |
| --- |
| **참고: 문자 코드**  컴퓨터 내부에서 문자는 이진수 숫자 형태로 저장된다.  따라서 컴퓨터에 저장될 수 있는 모든 문자에는, 그 문자를 저장하기 위한 숫자가 지정되어 있다.  각각의 문자에 숫자를 지정하는 방식을 문자 인코딩(character encoding) 이라고 한다.  (character => 문자, encoding => 이진수 코드화 한다)  여러 종류의 문자 인코딩이 있다. (예: Unicode, UTF-8, EUC-KR, ANSI, CP949 ...)  Java는 Unicode 문자 인코딩을 사용한다.  문자에 지정된 숫자를 문자 코드라고 한다.  charAtCode 메소드는, 지정된 위치의 문자의 Unicode 값을 리턴한다. |

### chatAt2.js

|  |
| --- |
| let s = "안녕하세요";  for (let i = 0; i < s.length; ++i)    console.log("%s %s %d", s[i], s.charAt(i), s.charCodeAt(i)); |

출력

|  |
| --- |
| 안 안 50504  녕 녕 45397  하 하 54616  세 세 49464  요 요 50836 |

## split 메소드

### 문자열.split(구분\_문자열)

문자열을 **구분\_문자열**을 기준으로 잘라서, 배열을 만들어서 리턴한다.

원본 문자열은 변경되지 않는다.

### split.js

|  |
| --- |
| let s = "one two three four";  let a = s.split(" ");  for (let i = 0; i < a.length; ++i)    console.log(a[i]); |

출력

|  |
| --- |
| one  two  three  four |

**구분\_문자열**에 정규식을 사용할 수 있다.

### split2.js

|  |
| --- |
| let s = "one, two,three  four";  let a = s.split(/[\t ,]+/);  for (let i = 0; i < a.length; ++i)    console.log(a[i]); |

출력

|  |
| --- |
| one  two  three  four |

/[\t ,]+/

탭, 공백, 콤마 문자들의 결합과 일치하는 정규식

### split3.js

|  |
| --- |
| let s = "hello";  let a = s.split("");  for (let i = 0; i < a.length; ++i)    console.log(a[i]); |

출력

|  |
| --- |
| h  e  l  l  o |

**구분\_문자열**이 빈문자열("")이면, 문자 한개씩 쪼개져 만들어진 배열을 리턴한다.

## match 메소드

### match1.js

|  |
| --- |
| let a = "list.jsp?pg=324&sz=15".match(/pg=[0-9]+/)  console.log(a) |

출력

|  |
| --- |
| [  'pg=324',  index: 9,  input: 'list.jsp?pg=324&sz=15',  groups: undefined  ] |

문자열의 match 메소드는,

문자열에서 정규식 패턴과 일치하는 부분을 찾아서,

일치하는 부분에 대한 정보를 배열로 리턴한다.

|  |  |
| --- | --- |
| 출력 항목 | 설명 |
| 'pg=324' | /pg=[0-9]+/ 정규식과 일치하는 문자열 |
| index: 9, | 일치하는 문자열의 위치 |
| input: 'list.jsp?pg=324&sz=15' | 입력 문자열 |

### match2.js

|  |
| --- |
| let a = "list.jsp?pg=324&sz=15".match(/pg=([0-9]+)/)  console.log(a) |

출력

|  |
| --- |
| [  'pg=324',  '324',  index: 9,  input: 'list.jsp?pg=324&sz=15',  groups: undefined  ] |

노란색으로 칠한 부분만, match1.js 코드와 다르다.

|  |  |
| --- | --- |
| 출력 항목 | 설명 |
| 'pg=324' | /pg=([0-9]+)/ 정규식과 일치하는 문자열 |
| 324 | 정규식에서 ( ) 괄호 부분에 일치하는 문자열 |
| index: 9, | 일치하는 문자열의 위치 |
| input: 'list.jsp?pg=324&sz=15' | 입력 문자열 |

# 요약

javascript 문자열은 " 문자로 묶어도 되고, ' 문자로 묶어도 된다.

문자열 내부에 " 문자가 포함되어 있다면, 그 문자열은 ' 문자로 묵어야 한다. '"hello world"'

javascript 문자열은 immutable 하다.

문자열이 생성된 후, 그 내용이 변경될 수 없다.

문자열 메소드들은 원본 문자열을 변경하는 것이 아니고, 새 문자열을 생성해서 리턴한다.

### 문자열.indexOf(부분\_문자열, 시작\_위치)

**문자열**에서 **부분\_문자열**을 찾아서, 찾은 위치(index)를 리턴한다.

**시작\_위치**에서부터 뒤쪽으로(순방향으로) 찾기 시작한다.

**시작\_위치** 파라미터가 생략된 경우 디폴트 값은 0 이다.

찾지 못할 경우 리턴값은 -1 이다.

### 문자열.lastIndexOf(부분\_문자열, 시작\_위치)

**문자열**에서 **부분\_문자열**을 찾아서, 찾은 위치(index)를 리턴한다.

**시작\_위치**에서부터 앞쪽으로(뒷방향으로) 찾기 시작한다.

**시작\_위치** 파라미터가 생략된 경우 디폴트 값은 문자열의 끝 위치이다.

찾지 못할 경우 리턴값은 -1 이다.

### 문자열.search(/정규식/)

**문자열**에서 **정규식**을 찾아서, 찾은 위치(index)를 리턴한다.

javascript 언어에서 정규식은 **/정규식/** 형태이다.

### 문자열.slice(시작\_위치, 끝\_위치)

문자열에서 **시작\_위치**에서 **끝\_위치** 사이의 부분 문자열을 리턴한다. (시작\_위치 <= 위치 < 끝\_위치)

원본 문자열은 변하지 않는다. (immutable)

**끝\_위치** 파라미터를 생략한 경우 디폴트 값은 문자열의 길이이다.

**시작\_위치** 값이 음수이면, 문자열 끝에서 역방향 인덱스이다.

### substring 메소드

substring 메소드는 slice 메소드와 유사하다.

차이점은, substring 메소드는 음수 파라미터를 사용할 수 없다는 점이다.

### substr 메소드

substr 메소드는 slice 메소드와 유사하다.

차이점은, substr 메소드의 두 번째 파라미터는 부분 문자열의 길이라는 점이다.

### 문자열.replace(부분\_문자열, 치환할\_문자열)

replace 메소드는, 문자열에서 **부분\_문자열**을 찾아서 **치환할\_문자열**로 치환한다.

일치하는 부분 문자열이 여러 개 있어도, 첫 번째 부분 문자열만 치환된다.

**부분\_문자열**에 정규식을 사용할 수 있다.

원본 문자열은 수정되지 않고, 치환된 새 문자열이 만들어져서 리턴된다.

toLowerCase 메소드 - 모든 문자를 소문자로 변경한 새 문자열을 리턴한다.

toUpperCase 메소드 - 모든 문자를 대문자로 변경한 새 문자열을 리턴한다.

### concat 메소드

파라미터 문자열들을 결합한 새 문자열을 리턴한다.

### trim 메소드

문자열 앞뒤의 공백을 제거한 새 문자열을 리턴한다.

### 문자열[위치]

문자열에서 위치(index)의 문자를 리턴한다.

### 문자열.chatAt(위치)

문자열에서 위치(index)의 문자를 리턴한다.

### 문자열.chatCodeAt(위치)

문자열에서 위치(index)의 문자의 코드 값을 리턴한다.

### 문자열.split(구분\_문자열)

문자열을 **구분\_문자열**을 기준으로 잘라서, 배열을 만들어서 리턴한다.

**구분\_문자열**에 정규식을 사용할 수 있다.

**구분\_문자열**이 빈문자열("")이면, 문자 한개씩 쪼개져 만들어진 배열을 리턴한다.

# 부록 - 정규식 (regular expression)

텍스트 에디터에서 문자열을 찾기 & 바꾸기 기능을 생각해 보자. (find & replace 기능)

Windodws의 메모장의 찾기 & 바꾸기 기능은, 찾을 문자열과 완전히 동일한 문자열만 찾을 수 있다.

예를 들어 찾을 문자열이 **the** 이면 정확히 **the** 만 찾아지고 **The** 는 찾아지지 않는다.

문자열을 찾을 때, 완전히 동일한 문자열만 찾는 것이 아니고, 어떤 패턴과 일치하는 문자열을 전부 다 찾을 수 있으면 편리할 것이다.

찾을 문자열의 패턴을 지정하는 표현식(expression)이 정규식(regular expression) 이다.

예를 들어서 **the** 뿐만 아니라 **The** 도 찾을 수 있기 위한 정규식은 **[tT]he** 이다.

eclipse, visual studio 등 대부분의 개발 도구는 정규식 찾기 기능을 지원한다.

## 정규식 특수 문자와 일반 문자

정규식에서 특수 문자가 아닌 평범한 문자는, 정확히 일치하는 것만 찾는다.

예를 들어 정규식 **[tT]he** 에서 **he** 부분은 평범한 문자이다. 그래서 he 부분은 정확히 일치해야 한다.

정규식에서 특수 문자는 찾을 패턴을 지정한다.

## [ ] 특수 문자

괄호 안의 문자들 중에서 아무 것이나 한 문자와 일치하는지 비교한다

예1) **[tT]he**

**[tT]** 부분은 **t** 문자와도 일치하고 **T** 문자와도 일치한다.

**th** 부분은 특수 문자가 아니기 때문에 정확히 일치해야 한다.

그래서 **[tT]he** 정규식은 **the** 와도 일치하고 **The** 와도 일치한다.

예2) **9[123][0123]**

**9** 부분은 특수 문자가 아니기 때문에 정확히 일치해야 한다.

일치하는 문자열들: 910 911 912 913 920 921 922 923 930 931 932 933

일치하지 않는 문자열들: 900 901 902 903 914 915 916 833

예3) **[123][0-3]8**

**8** 부분은 특수 문자가 아니기 때문에 정확히 일치해야 한다.

**[0-3]** 부분은 **[0123]** 과 동일하다.

따라서 일치하는 문자열들도 **[123][0123]8** 정규식과 동일하다.

예4) **[1-9][0-9]a**

**a** 부분은 특수 문자가 아니기 때문에 정확히 일치해야 한다.

**[1-9][0-9]** 정규식은 **[123456789][0123456789]** 정규식과 동일하다.

일치하는 문자열들: 12a 23a 35a 78a 90a 23a 30a 98a 81a 74a 99a

일치하지 않는 문자열들: 00a 01a 02a 03a 04a 05a 06a 07a 08a 09a 99b

예5) **0x[0-9A-F][0-9A-F]**

**0x[0-9A-F][0-9A-F]** 정규식은 **0x[0123456789ABCDEF][0123456789ABCDEF]** 정규식과 동일하다.

**0x** 부분은 특수 문자가 아니기 때문에 정확히 일치해야 한다.

일치하는 문자열들: 0x2F 0x34 0xAB 0xEE

일치하지 않는 문자열들: x2F 0x3f

## . 특수 문자

아무 문자 한 개와 일치한다.

예1) **A.B**

일치하는 문자열: AaB AzB A3B A@B A#B

일치하지 않는 문자열: AB AacB AxyzB

예1) **A..B**

일치하는 문자열: AabB AxzB A3aB A4@B A!#B

일치하지 않는 문자열: AB AaB AxyzB

## [^ ] 특수 문자

괄호 안의 문자들을 제외한 다른 문자들 중에서 아무 것이나 한 문자와 일치하는지 비교한다

예1) **[^tT]he**

일치하는 문자열: ahe bhe che Ahe Bhe Che 3he #he @he

일치하지 않는 문자열: the The

예2) **[^0-5][A-F]**

**[^0-5][A-F]** 정규식은 **[^012345][ABCDEF]** 정규식과 동일하다.

일치하는 문자열: 6A 7B 8A 9F AC aB bF

일치하지 않는 문자열: 0A 1B 2C 5F af bH 6Z

## ? 특수 문자

? 특수 문자 바로 앞의 정규식과 일치하는 문자 0 ~ 1 개 있는지 비교한다.

예1) **AB?C**

A C 부분은 정확히 일치해야 한다.

B? 부분은 B 문자가 0 ~ 1 개만 있는지 비교한다.

일치하는 문자열: AC ABC

일치하지 않는 문자열: ABBC ADC BBC

예2) **A[d-f]?C**

A C 부분은 정확히 일치해야 한다.

[d-f]? 부분은 [d-f] 정규식과 일치하는 문자가 0 ~ 1 개만 있는지 비교한다.

일치하는 문자열: AC AdC AeC AfC

일치하지 않는 문자열: AddC AEC

## \* 특수 문자

\* 특수 문자 바로 앞의 정규식과 일치하는 문자 0 ~ 여러 개 있는지 비교한다.

예1) **AB\*C**

A C 부분은 정확히 일치해야 한다.

B\* 부분은 B 문자가 0 ~ 여러 개 있는지 비교한다.

일치하는 문자열: AC ABC ABBC ABBBC ABBBBC

일치하지 않는 문자열: ACC ADC BBC

예2) **A[d-f]\*C**

A C 부분은 정확히 일치해야 한다.

[d-f]\* 부분은 [d-f] 정규식과 일치하는 문자가 0 ~ 여러 개 있는지 비교한다.

일치하는 문자열: AC AdC AfeC AdeeC

일치하지 않는 문자열: AdgC AbeC AEC

## + 특수 문자

+ 특수 문자 바로 앞의 정규식과 일치하는 문자 1 ~ 여러 개 있는지 비교한다.

예1) AB+C

A C 부분은 정확히 일치해야 한다.

B+ 부분은 B 문자가 1 ~ 여러 개 있는지 비교한다.

일치하는 문자열: ABC ABBC ABBBC ABBBBC

일치하지 않는 문자열: AC CC ADC BBC

예2) A[d-f]+C

A C 부분은 정확히 일치해야 한다.

[d-f]+ 부분은 [d-f] 정규식과 일치하는 문자가 1 ~ 여러 개 있는지 비교한다.

일치하는 문자열: AdC AeC AfC AdefC AeeffC

일치하지 않는 문자열: AC AEC ADEC A3378C

## ^ 특수 문자

^ 특수 문자는 줄바꿈 문자로 구분되는 각 줄의 선두 부분에서 있는 문자열인지 검사한다.

예1) ^123

줄의 선두 부분에 있는 123 문자열과 일치한다.

줄의 선두가 아니면 일치하지 않는다.

## $ 특수 문자

$ 특수 문자는 줄바꿈 문자로 구분되는 각 줄의 끝 부분에서 있는 문자열인지 검사한다.

예1) abc$

줄의 끝 부분에 있는 abc 문자열과 일치한다.

줄의 끝 부분이 아니면 일치하지 않는다.

## {*n*} 특수 문자

{ } 괄호 안에는 숫자가 1 개 들어 있어야 한다.

{ } 괄호 바로 앞의 정규식과 일치하는 문자가 반복되는 수가, { } 괄호 안의 숫자와 일치해야 한다.

예1) AB{3}C

일치하는 문자열: ABBBC

예2) A[0-3]{4}C

일치하는 문자열: A0000C A0123C A0023C A1320C A3211C

## {*n,m*} 특수 문자

{ } 괄호 안에는 숫자가 2 개 들어 있어야 한다.

{ } 괄호 바로 앞의 정규식과 일치하는 문자가 반복되는 수가, n 보다 크거나 같고 m 보다 작거나 같아야 한다.

예1) AB{2,3}C

일치하는 문자열: ABBC ABBBC

예2) A[0-3]{2,4}C

일치하는 문자열: A00C A013C A0023C A3211C

## ( ) 특수 문자

( ) 괄호 안의 정규식을 묶는 역할을 한다.

예1) A(BC)?D

일치하는 문자열: AD ABCD

예2) A(BC){2,3}D

일치하는 문자열: ABCBCD ABCBCBCD

예3) A([a-c][0-3]){2}D

일치하는 문자열 Aa0b2D Ac2b0D Ab3c2D

일치하지 않는 문자열 Aba20D

예4) A(23)+D

일치하는 문자열: A23D A2323D A232323D

## | 특수 문자

| 특수 문자 앞 뒤의 정규식 둘 중의 하나와 문자열이 일치해야 한다.

예1) Ab|cD

| 특수 문자 앞의 정규식은 b 이고 뒤의 정규식은 c 이다.

위 정규식은 A[bc]D 정규식과 동일하다.

일치하는 문자열: AbD AcD

예2) A(bc)|(def)G

| 특수 문자 앞의 정규식은 ac 이고 뒤의 정규식은 def 이다.

일치하는 문자열: AbcG AdefG

## \ 특수 문자

정규식 특수 문자를 보통의 문자처럼 비교할 때, 특수 문자 앞에 \ 문자를 붙인다.

예1) Ab\?D

일치하는 문자열: Ab?D

예2) A[0-3]\+D

일치하는 문자열: A0+D A1+D A2+D A3+D

예3) \([1-9][0-9]\*\)

일치하는 문자열: (1) (2) (3) (10) (21) (34) (100) (235)

일치하지 않는 문자열: (0) (01) [10]

예4) \[1-9\]

일치하는 문자열: [1-9]

예5) [\t ]

탭 문자, 공백 문자 중의 한 문자와 일치한다.

예6) [\t ]+

탭 문자나 공백 문자가 1 개 이상 반복된 부분과 일치한다.

공백 문자 사이 사이에 탭 문자가 있어도 일치하고

탭 문자만 여러 개 반복되어도 일치하고

탭 문자 한 개나 공백 문자 한 개와도 일치한다.

## 연습문제

다음 정규식과 일치하는 문자열의 예를 적어라.

[a-zA-Z\_][a-zA-Z0-9\_]\*

[0-9]{6}-[1234][0-9]{6}

01[0-9]-[0-9]{3,4}-[0-9]{4}

\([1-9][0-9]\*\)

0x[0-9A-F]{2}

"%[0-9]\*[dsfx]"

[a-zA-Z]+@[a-zA-Z]+\.com

# 과제

## 모음 위치 (vowel.js)

"When I find myself in times of trouble Mother Mary comes to me Speaking words of wisdom, let it be."

위 문자열에서 모음(aeiou)들의 위치를 출력하는 코드를 구현하시오.

## 공백 제거 (space.js)

위 문제의 문자열에서 공백을 모두 제거한 새 문자열을 만들고,

그 문자열을 출력하시오.

## 쪼개기 (split.js)

위 문제의 문자열에서 공백을 기준으로 문자열을 쪼개서, 문자열 배열을 만들고,

그 배열을 출력하시오.